Chapter 3

Methodology

In this chapter, we present our methodology to answer the research questions enunciated in **??**. We investigate three research questions. In the first question, we artificially generate WebAssembly program variant, and we qualitatively compare the ability of CROW, one of the contributions of this thesis, to generate statically different yet functionally equivalent variants. Our second research question statically compares the generated variants and compares their behavior through traces, execution time, and preservation through the machine code compiling process. The final research question evaluates the feasibility of using the program variants in security-sensitive environments such as Edge-Cloud computing proposing a multivariant execution approach.

We share the same corpora of programs to answer all our research questions. We first enunciate the corpora of programs. Then, for each research question, we establish the metrics, set the configuration for the experiments, and describe the protocol for them.

3.1 Corpora

Our experiments assess the impact of artificially created diversity. For such reason, the first step is to build a suitable corpus of programs to generate program variants. We then use the generated variants to study their static, dynamic, and security properties. We answer all our research questions with three corpora of programs appropriate for our experiments. In Table 3.1 we listed the corpus name, the number of programs inside the corpus, the total number of functions, the range of lines of code, and the original location of the corpus. In the following, we describe the filtering and description of each corpus.

1. **Rosetta** corpus is part of the CROW contribution []. We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for

[1]`http://www.rosettacode.org/wiki/Rosetta_Code`

specific programming tasks in various programming languages. It contains a wide range of tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from Rosetta Code, which represents 989 programs as of 01/26/2020. We then apply a number of filters: the programs should successfully compile, they should not require user inputs, the programs should terminate and should not provide in non-deterministic results. The result of the filtering is a corpus of 303 C programs. All programs have a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

2. **Libsodium** is part of both CROW and MEWE contributions [] []. This project is an encryption, decryption, signature and password hashing library which can be ported to WebAssembly. We selected 5 programs or endpoints to answer our research questions. These endpoints have between 8 and 2703 lines of code per function. The project is selected based on their suitability for diversity synthesis with CROW, *i.e.,* the project should have the ability to collect its modules in LLVM intermediate representation and the project should be easily portable Wasm/WASI.

3. **QrCode** is part of the MEWE contribution. This project is a QrCode and MicroQrCode generator written in Rust. We selected 2 programs or endpoints to answer our research questions. These endpoints have between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for diversity synthesis with CROW.

We build our corpora in an escalating strategy. The first corpus should illustrate the feasibility of CROW to generate program variants out of simple programs in terms of code size. The latter two corpora study the impact of CROW on more extensive real-world programs, including one project meant for security-sensitive applications. Overall, all corpora are considered to come along the LLVM pipeline, having the input for CROW from C/C++ source code or LLVM bitcodes. We base this decision on the previous experimental work of Lehman et al. [?] . This work shows that more than 70% of all WebAssembly programs come out of LLVM based tooling.

3.2 RQ1. To what extent we can generate program variants for WebAssembly?

This research question investigates whether we can artificially generate program variants for WebAssembly. Concretely, we use CROW to generate program variants from an original program, written in C/C++ or directly passing an LLVM bitcode module to it. Therefore, the first step and research question is related to the ability of CROW to generate a handful number of program variants. Our intuition is

Corpus name	No. programs	No. functions	LOC range	Location
Rosetta	303	303	7 - 150	`http://rosettacode.org/wiki/Rosetta_Code`
Libsodium	5	869	8 - 2703	`https://github.com/jedisct1/libsodium`
QrCode	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust`
Total	310	3021		

Table 3.1: Corpora description. The table is composed by the name of the corpus, the number of programs, the number of functions, the lines of code range and the location of the corpus.

that the larger the number of generated variants, the better the program variants' security and reliability properties can offer.

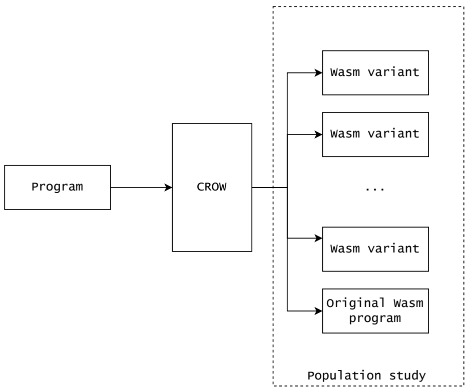

Figure 3.1: Simplification of the program variants generation workflow.

In Figure 3.1 we simplify the workflow to generate WebAssembly program variants. We pass each function of the corpora to CROW. To answer RQ1, we study the outcome of this pipeline, the generated variants.

3.2.1 Metrics

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of unique variants generated by CROW for each original function of each corpus.

Metric 1 *Population size $S(P)$: Given a program P and its generated variants V, the population size metric is defined as.*

$$S(P) = |V|$$

Notice that, the variant population includes P as an instance.

A program and its variants compose what we call a program's population. Notice that all proposed metrics over programs and their variants make sense only at the population level. Therefore, it only makes sense to compare semantically equivalent programs, i.e., from the same population. Along with this work, we use the term "program's population" to refer to a program and its variants.

3.2.2 Protocol

One design property of CROW is that all possible programs that can be generated for a given language (LLVM in this case) are constructed and verified for equivalence. Thus, there are two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all the 60 supported instructions in the synthesizer.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus, we set the exploration time to 1 hour maximum per function for Rosetta . In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function in the exploration stage. The decision behind the usage of lower timeout for Libsodium and Libsodium is motivated by the properties listed in Table 3.1. The latter two corpora are remarkably larger in terms of the number of instructions and functions count.

We pass each of the 303 + 869 + 1849 functions in the corpora to CROW, as Figure 3.1 illustrates, to synthesize program variants. We then calculate Metric 1 for each program's population and conclude by answering RQ1.

3.3 RQ2. To what extent the artifically generated variants are different?

In this second research question, we investigate to what extent the artificially created variants are different between them and to the original program. To conduct this research question, we perform three main experiments as Figure 3.2 illustrates: static comparison, dynamic comparison and variant's preservation. We use the original functions from corpora described in section 3.1 and their variants generated in the answering of RQ1.

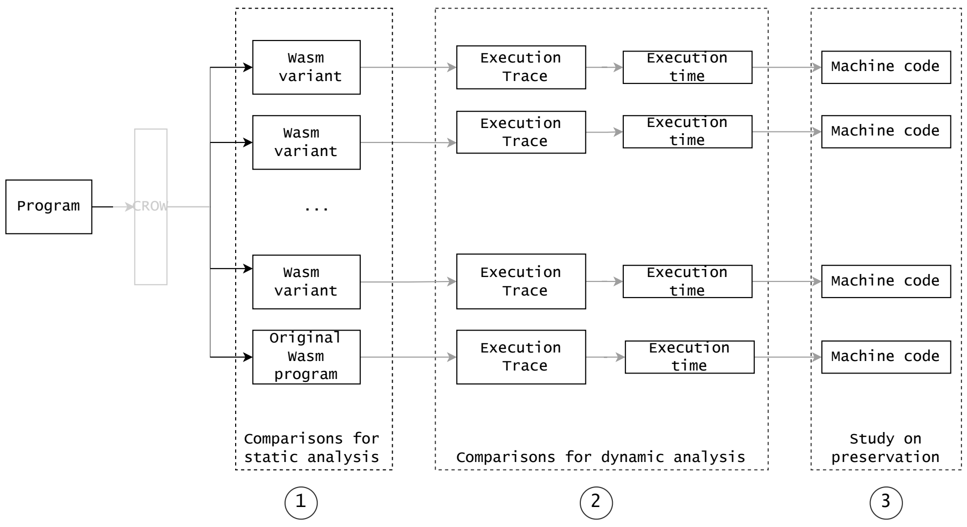

Figure 3.2: Population study methodology for each original corpora and the variants generated in RQ1.

We first start by statically comparing all program's populations generated in RQ1 from the original program of the corpora in Table 3.1. We execute each program on each population to collect their execution traces and execution times. Furthermore, we then compare their dynamic behavior. Finally, we compare the preservation of the variants after state-of-the art WebAssembly engines compile them.

To conclude on the program's population, we perform fine-grained comparisons by studying all pairs of programs in the population. Therefore, all defined metrics are formulated to support a pairwise comparison strategy. In the following, we define the metrics used to answer RQ2.

3.3.1 Metrics

To compare two programs of the same population statically, we propose a global alignment approach using Dynamic Time Warping (DTW). In previous work, we highlighted how this approach measure similarity [?] . Dynamic Time Warping [?] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. In the following, we define the *dt_static* metric.

Metric 2 *dt_static: Given two programs of the same program's population P_X and V_X written in X code, dt_static(P_X, V_X), computes the DTW distance between the corresponding program instructions for representation X.*

A dt_static(P_X, V_X) of 0 *means that the code of both the original program and the variant is the same, i.e., they are statically identical in the representation X. The higher the value of dt_static, the more different the programs are in representation X.*

Notice that, for comparing WebAssembly programs, the metric is the instantiation of dt_static with $X = WebAssembly$.

We measure the difference between programs at runtime by comparing their execution times and execution traces. We compare their execution traces with an alignment metric at the function and instruction level.

Metric 3 *dt_dyn: Given a program P, a CROW generated variant P' and T a trace space ($T \in \{Function, Instruction\}$) dt_dyn(P,P',T), computes the DTW distance between the traces collected during their execution in the T space. A dt_dyn of* 0 *means that both traces are identical.*

The higher the value, the more different the traces.

Metric 4 *Execution time: Given a WebAssembly program P, the execution time is the time spent to execute the binary.*

WebAssembly is an intermediate language, and interpreters produce machine code to execute them. For program variants, this means that compiling can undo artificially introduced transformations, for example, through optimization passes. When a code transformation for a variant is maintained from the first time it is introduced to the final machine, code generation is then a preserved variant. For this mentioned reasoning, we need a preservation metric. The critical property we consider is as follows:

Property 1 *Preservation: Given a program P and a variant V' from the same program's population, if dt_static(P_{Wasm}, P'_{Wasm}) > 0 and dt_static(P_{x86}, P'_{x86}) $> 0 \implies$ both programs are still different when compiled to machine code.*

If the property fits for two programs, then the underlying compiler does not remove the transformations made by CROW.

Metric 5 *Preservation: Given a WebAssembly programs P and a collection of generated variants V, the preservation ratio is the number of pair of programs that fit with Property 1 for a specific compiling engine over the total number of program pairs.*

3.3.2 Protocol

For each program's population generated in answering RQ1, we compare the sequence of instructions of each variant with the initial program and the other variants. We obtain the Metric 2 values for each program-variant WebAssembly pair code.

To compare program and variants behavior during runtime, we analyze all the unique program variants generated by CROW in a pairwise comparison. We use SWAM[2] to execute each program and variant to collect the function and instruction traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions, including the virtual stack operations.

Furthermore, we collect the execution time, Metric 4, for all programs and their variants. We execute each program or variant 10000 times, and we compare the collected execution time distributions using a Mann-Withney test [?] in a pairwise strategy.

We collect Metric 5 for all program pairs in all program populations. We use two WebAssembly engines to study variant's preservation after compiling.

- **V8** [?] : the engine used by Chrome and NodeJS to execute JavaScript and WebAssembly.
- **wasmtime** [?] : a standalone runtime for WebAssembly. This engine is used by the Fastly platform to provide Edge-Cloud computing services.

We only consider the x86 representation after the WebAssembly code is compiled to the machine code. This decision is not arbitrary. According to a previous study [?] , any conclusion carried out by comparing two program binaries under a specific target can be extrapolated to another target for the same binaries.

3.4 RQ3. To what extent the artificially generated variants can be used to enforce security on Edge-Cloud platforms?

In the last research question, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant execution binaries. For this purpose, we build multivariant binaries to be deployed at the Edge in the Fastly platform. We use the variants generated for the programs of the Libsodium and QrCode corpora, $2 + 5$ programs involving $869 + 1849$ functions respectively. Multivariant binaries are created by converting each program's population into a single function for which each call at runtime selects and executes a different variant. One of the contributions of this work

[2]`https://github.com/satabin/swam`

is MEWE, a tool that automatically creates multivariant binaries out of program variants generated by CROW. We simplify the protocol to answer RQ3 in Figure 3.3.

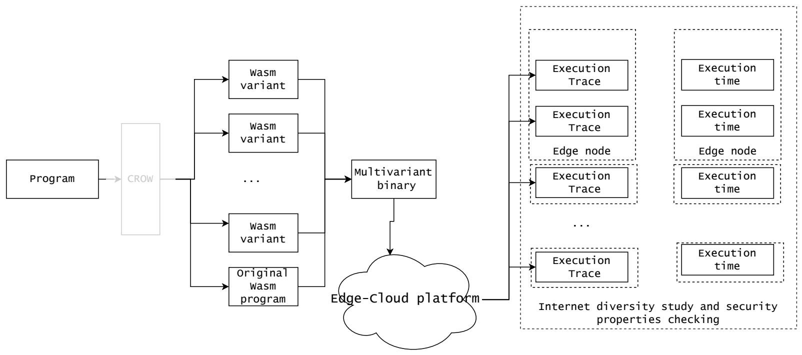

Figure 3.3: Multivariant binary creation and workflow for RQ3.

We assess the ability of MEWE to produce binaries that exhibit random execution paths when executed on the edge. We check the diversity of execution traces gathered from the execution of a multivariant binary. The traces are collected from all edge nodes to assess Multivariant Execution (MVE) worldwide. MEWE generates binaries that embed a multivariant behavior. Finally, we measure how MEWE generates different execution times on the edge. Then, we discuss how multivariant binaries contribute to less predictable timing side-channels.

3.4.1 Metrics

To compare the diversity of function traces for the 7 created multivariant binaries, we enunciate the following metrics.

Metric 6 *Unique traces: $R(n,e)$. Let $S(n,e) = \{T_1, T_2, ..., T_{100}\}$ be the collection of 100 traces collected for one endpoint e on an edge node n, $H(n,e)$ the collection of hashes of each trace and $U(n,e)$ the set of unique trace hashes in $H(n,e)$. The uniqueness ratio of traces collected for edge node n and endpoint e is defined as*

$$R(n,e) = \frac{|U(n,e)|}{|H(n,e)|}$$

Metric 7 *Normalized Shannon entropy: $E(e)$ Let e be an endpoint, $C(e) = \cdot_{n=0}^{64} H(n,e)$ be the union of all trace hashes for all edge nodes. The normalized Shannon Entropy for the endpoint e over the collected traces is defined as:*

$$E(e) = -\Sigma \frac{p_x * log(p_x)}{log(|C(e)|)}$$

Where p_x is the discrete probability of the occurrence of the hash x over $C(e)$.

Notice that we normalize the standard definition of the Shannon Entropy, Metric 7, by using the perfect case where all trace hashes are different. This normalization allows us to compare the calculated entropy between endpoints. The value of the metric can go from 0 to 1. The worst entropy, value 0, means that the endpoint always perform the same path independently of the edge node and the number of times the trace is collected for the same node. On the contrary, 1 for the best entropy, when each edge node executes a different path every time the endpoint is requested.

3.4.2 Protocol

We run the experiments to answer RQ3 on the Fastly edge computing platform. We deploy and execute the original and the multivariant endpoints on 64 edge nodes located around the world[3]. These edge nodes usually have an arbitrary and heterogeneous composition in architecture and CPU model.

We execute each endpoint multiple times on each node to measure the diversity of execution traces exhibited by the multivariant binaries. Each query on the same endpoint is performed with the same input value. This guarantees that if we observe different traces for different executions, it is due to the presence of multiple function variants. The inputs that we pass to execute the endpoints at the edge and the received output for all executions are available in the reproduction repository at **TODO** REPO .

For each query, we collect the execution trace, i.e., the sequence of function names that have been executed when triggering the query. We instrument the multivariant binaries to record each function entrance to observe these traces.

We then measure the number of unique execution traces exhibited by each multivariant binary, Metric 6, on each separate edge node. Then, to compare the traces, we hash them with the `md5sum` function. We then calculate the number of unique hashes among the 100 traces collected for an endpoint on one edge node. We follow by collecting the normalized Shannon entropy, Metric 7, for all collected execution traces for each endpoint. The Shannon Entropy gives the uncertainty in the outcome of a sampling process. If a specific trace has a high frequency of appearing in part of the sampling, then it is inevitable that this trace will appear in the other part of the sampling.

We calculate Metric 7 for the 7 endpoints, for 100 traces collected from 64 edge nodes, for a total of 6400 collected traces per endpoint. Each trace is collected in a round-robin strategy, i.e., the traces are collected from the 64 edge nodes sequentially. For example, we collect the first trace from all nodes before continuing to collect the second trace. This process is followed until 100 traces are collected from all edge nodes.

[3]The number of nodes provided in the whole platform is 72, we decided to keep only the 64 nodes that remained stable during our experimentation.

In addition, we collect 100k execution times for each binary, both the original and multivariant binaries. We perform a Mann-Withney U test [**?**] to compare both execution time distributions. If the P-value is lower than 0.05, two compared distributions are different.

3.5 Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of CROW to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through static analysis, dynamic analysis, and a variant's preservation study. Finally, we propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Nevertheless, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for WebAssembly programs. In the next chapter, we present and discuss the results obtained with this methodology.

Appended papers